

DOCKET NO.: MSFT-0323/167389.01
Application No.: 09/838,436

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:

**Istvan Cseri, Oliver Nicolas Seeliger,
Andrew J. Layman**

Confirmation No.: 2254

Application No.: 09/838,436

Group Art Unit: 2144

Filing Date: April 19, 2001

Examiner: Melvin H. Pollack

For: Method and System for Providing an XML Binary Format

**EXPRESS MAIL LABEL NO:
DATE OF DEPOSIT:**

Mail Stop AF
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

DECLARATION PURSUANT TO 37 C.F.R. 1.131

I, Oliver N. Seeliger, declare and say that:

1. I am one of the named inventors of the invention described and claimed in U.S. Patent Application No. 09/838,436 (the above-identified application, hereinafter "the '436 application"), which was filed with the United States Patent and Trademark Office on April 19, 2001.

2. As an inventor, I am familiar with the '436 application and the associated rejection alleged in the outstanding Official Action, dated June 12, 2006. I am also familiar with the reference cited by the United States Patent and Trademark Office (Girardot, Marc and Sundaresan, Neel, "Millau: an encoding format for efficient representation and exchange of XML over the Web," hereinafter "Girardot et al.") in connection with the latest Official Action.

3. I understand that the publishing date for Girardot et al. is sometime in mid-2000, as is documented in Appendix A. In one of the Office Actions, dated December 15,

Appendix A: Publication Date of Girardot et al. reference

Source: <http://www.informatik.uni-trier.de/~ley/db/journals/cn/cn33.html>

dblp.uni-trier.de

I. Computer Networks, Volume 33

A. Volume 33, Numbers 1-6, June 2000

Ninth International World Wide Web Conference Performance

- Nina T. Bhatti, Anna Bouch, Allan Kuchinsky:
Integrating user-perceived quality into Web server design. 1-16
Electronic Edition (link) [BibTeX](#)
- Balachander Krishnamurthy, Craig E. Wills:
Analyzing factors that influence end-to-end Web performance. 17-32
Electronic Edition (link) [BibTeX](#)
- Pablo Rodriguez, Sandeep Sibal:
SPREAD: Scalable platform for reliable and efficient automated distribution. 33-49
Electronic Edition (link) [BibTeX](#)

Security

- David P. Kormann, Aviel D. Rubin:
Risks of the Passport single signon protocol. 51-58
Electronic Edition (link) [BibTeX](#)
- Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati:
Design and implementation of an access control processor for XML documents. 59-75
Electronic Edition (link) [BibTeX](#)
- Brant Hashii, Scott Malabarba, Raju Pandey, Matt Bishop:
Supporting reconfigurable security policies for mobile programs. 77-93
Electronic Edition (link) [BibTeX](#)

Rich Queries

- Ellen Spertus, Lynn Andrea Stein:
Squeal: a structured query language for the Web. 95-103
Electronic Edition (link) [BibTeX](#)
- Thomas A. Phelps, Robert Wilensky:
Robust intra-document locations. 105-118
Electronic Edition (link) [BibTeX](#)
- Daniela Florescu, Donald Kossmann, Ioana Manolescu:
Integrating keyword search into XML query processing. 119-135
Electronic Edition (link) [BibTeX](#)

XML 1

- Stefano Ceri, Piero Fraternali, Aldo Bongio:
Web Modeling Language (WebML): a modeling language for designing Web sites. 137-157
Electronic Edition (link) [BibTeX](#)
- Greg J. Badros:
JavaML: a markup language for Java source code. 159-177
Electronic Edition (link) [BibTeX](#)
- Patrick van Amstel, Pim van der Eijk, Evert Haasdijk, David Kuilman:
An interchange format for cross-media personalized publishing. 179-195
Electronic Edition (link) [BibTeX](#)

Mobile Access

- Masahiro Hori, Goh Kondoh, Kouichi Ono, Shin'ichi Hirose, Sandeep K. Singhal:
Annotation-based Web content transcoding. 197-211
Electronic Edition (link) [BibTeX](#)
- Orkut Buyukkokten, Hector Garcia-Molina, Andreas Paepcke:
Focused Web searching with PDAs. 213-230
Electronic Edition (link) [BibTeX](#)
- Eija Kaasinen, Matti Aaltonen, Juha Kolari, Suvi Melakoski, Timo Laakko:
Two approaches to bringing Internet services to WAP devices. 231-246
Electronic Edition (link) [BibTeX](#)

Storage Infrastructure

- Raymie Stata, Krishna Bharat, Farzin Maghoul:
The Term Vector Database: fast access to indexing terms for Web pages. 247-255
Electronic Edition (link) [BibTeX](#)
- Brian E. Brewington, George Cybenko:
How dynamic is the Web? 257-276
Electronic Edition (link) [BibTeX](#)
- Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, Andreas Paepcke:
WebBase: a repository of Web pages. 277-293
Electronic Edition (link) [BibTeX](#)

Web Characterization

- Monika Rauch Henzinger, Allan Heydon, Michael Mitzenmacher, Marc Najork:
On near-uniform URL sampling. 295-308
Electronic Edition (link) [BibTeX](#)
- Andrei Z. Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, Janet L. Wiener:
Graph structure in the Web. 309-320
Electronic Edition (link) [BibTeX](#)
- Anja Feldmann:
BLT: Bi-Layer Tracing of HTTP and TCP/IP. 321-335
Electronic Edition (link) [BibTeX](#)

Practice and Experience

- Christoph Hölscher, Gerhard Strube:
Web search behavior of Internet experts and newbies. 337-346
Electronic Edition (link) [BibTeX](#)
- Kerstin Forsberg, Lars Dannstedt:
Extensible use of RDF in a business context. 347-364
Electronic Edition (link) [BibTeX](#)
- Shannon L. Fowler, Anne-Marie J. Novack, Michael J. Stillings:
The evolution of a manufacturing Web site. 365-376
Electronic Edition (link) [BibTeX](#)

Hypermedia and Link Analysis

- Ramesh Sarukkai:
Link prediction and path analysis using Markov chains. 377-386
Electronic Edition (link) [BibTeX](#)
- Ronny Lempel, Shlomo Moran:
The stochastic approach for link-structure analysis (SALSA) and the TKC effect. 387-401
Electronic Edition (link) [BibTeX](#)
- Harald Weinreich, Winfried Lamersdorf:
Concepts for improved visualization of Web link attributes. 403-416
Electronic Edition (link) [BibTeX](#)

Search and Ranking

- Atsushi Sugiura, Oren Etzioni:
Query routing for Web search engines: architecture and experiments. 417-429
Electronic Edition (link) [BibTeX](#)
- Lieming Huang, Matthias Hemmje, Erich J. Neuhold:
ADMIRE: an adaptive data model for meta search engines. 431-448
Electronic Edition (link) [BibTeX](#)

- Dell Zhang, Yisheng Dong:
An efficient algorithm to rank Web resources. 449-455
Electronic Edition (link) [BibTeX](#)
- Sougata Mukherjea:
WTMS: a system for collecting and analyzing topic-specific Web information. 457-471
Electronic Edition (link) [BibTeX](#)

Tools

- Steffen Staab, Jürgen Angele, Stefan Decker, Michael Erdmann, Andreas Hotho, Alexander Maedche, Hans-Peter Schnurr, Rudi Studer, York Sure:
Semantic community Web portals. 473-491
Electronic Edition (link) [BibTeX](#)
- Krishna Bharat:
SearchPad: explicit capture of search context to support Web search. 493-501
Electronic Edition (link) [BibTeX](#)
- Vinod Anupam, Juliana Freire, Bharat Kumar, Daniel F. Liewen:
Automating Web navigation with the WebVCR. 503-517
Electronic Edition (link) [BibTeX](#)
- John C. Shafer, Rakesh Agrawal:
Continuous querying in database-centric Web applications. 519-531
Electronic Edition (link) [BibTeX](#)

Hypermedia

- Sami Rollins, Neel Sundaresan:
AVoN calling: AXL for voice-enabled Web navigation. 533-551
Electronic Edition (link) [BibTeX](#)
- Kaj Grønbaek, Lennert Sloth, Niels Olof Bouvin:
Open hypermedia as user controlled meta data for the Web. 553-566
Electronic Edition (link) [BibTeX](#)
- Pengkai Pan, Glorianna Davenport:
I-Views: a community-oriented system for sharing streaming video on the Internet. 567-581
Electronic Edition (link) [BibTeX](#)
- Stuart Goose, Mike Newman, Claus Schmidt, Laurent Hue:
Enhancing Web accessibility via the Vox Portal and a Web-hosted dynamic HTMLVoxML converter. 583-592
Electronic Edition (link) [BibTeX](#)

Architecture

- Mike Spreitzer, Bill Janssen:
HTTP 'Next Generation'. 593-607
Electronic Edition (link) [BibTeX](#)

- Venky Krishnan, S. Grace Chang:
Customized Internet radio. 609-618
Electronic Edition (link) [BibTeX](#)
- Andrew C. Huang, Benjamin C. Ling, John J. Barton, Armando Fox:
Running the Web backwards: appliance data services. 619-631
Electronic Edition (link) [BibTeX](#)

Caching

- Kun-Lung Wu, Philip S. Yu:
Latency-sensitive hashing for collaborative Web caching. 633-644
Electronic Edition (link) [BibTeX](#)
- Jaeyeon Jung, Dongman Lee, Kilnam Chon:
Proactive Web caching with cumulative prefetching for large multimedia data. 645-655
Electronic Edition (link) [BibTeX](#)
- Stephane Gruber, Jennifer Rexford, Andrea Basso:
Protocol considerations for a prefix-caching proxy for multimedia streams. 657-668
Electronic Edition (link) [BibTeX](#)

Data Mining

- Soumen Chakrabarti, Sandeep Srivastava, Mallela Subramanyam, Mitul Tiwari:
Using Memex to archive and mine community Web browsing experience. 669-684
Electronic Edition (link) [BibTeX](#)
- William W. Cohen, Wei Fan:
Web-collaborative filtering: recommending music by crawling the Web. 685-698
Electronic Edition (link) [BibTeX](#)
- Neel Sundaresan, Jeonghee Yi:
Mining the Web for relations. 699-711
Electronic Edition (link) [BibTeX](#)

XML 2

- Alison Cawsey:
Presenting tailored resource descriptions: will XSLT do the job? 713-722
Electronic Edition (link) [BibTeX](#)
- Mary F. Fernandez, Wang Chiew Tan, Dan Suciu:
SilkRoute: trading between relations and XML. 723-745
Electronic Edition (link) [BibTeX](#)
- Marc Girardot, Neel Sundaresan:
Millau: an encoding format for efficient representation and exchange of XML over the Web. 747-765
Electronic Edition (link) [BibTeX](#)

E-Commerce

- John A. Tomlin:
An entropy approach to unintrusive targeted advertising on the Web. 767-774
Electronic Edition (link) [BibTeX](#)
- Naoki Abe, Tomonari Kamba:
A Web marketing system with automatic pricing. 775-788
Electronic Edition (link) [BibTeX](#)
- José Carlos Brustoloni, Juan A. Garay:
MicroISPs: providing convenient and low-cost high-bandwidth Internet access. 789-802
Electronic Edition (link) [BibTeX](#)

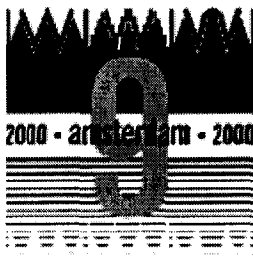
Content and Control

- Kirk Martinez, Steve T. Perry, John Cupitt:
Object browsing using the Internet Imaging Protocol. 803-810
Electronic Edition (link) [BibTeX](#)
- Takehiro Nakayama, Hiroki Kato, Yohei Yamane:
Discovering the gap between Web site designers' expectations and users' behavior. 811-822
Electronic Edition (link) [BibTeX](#)
- Davood Rafiei, Alberto O. Mendelzon:
What is this page known for? Computing Web page reputations. 823-835
Electronic Edition (link) [BibTeX](#)

DBLP: [[Home](#) | [Search: Author, Title](#) | [Conferences](#) | [Journals](#)]

Copyright © Thu Feb 8 20:07:13 2007 by [Michael Ley](#) (ley@uni-trier.de)

<http://www9.org:>



9th International World Wide Web Conference

THE WEB: THE NEXT GENERATION

II. Amsterdam, May 15 - 19, 2000

-
- Conference Organizers
 - Program Information
 - Conference Sponsors
 - Conference Exhibitors

Post Conference
Conference Paper Proceedings ■ Poster Proceedings

Developers' Day Presentations: XML Track

- Articles
- Past Conferences
- Volunteers Info
- Contact Us

III. Conference Topics

- Performance
- Security
- E-Commerce
- Querying
- XML
- Mobile
- Storage
- Measurement
- Modeling
- Searching
- Hypermedia
- Architecture
- Caching
- Data Mining
- Tools
- Online Learning
- Accessibility
- Collaboration
- Legal Issues
- Metadata

Culture Track Updated Info: Cultural Heritage and ICT

Order Conference Materials

Conference Awards

Keynote Speakers

Lawrence Lessig, Harvard Law School

- "Cyberspace's Constitution" (presentation)

Charles W. Davies, Psion PLC

- "Wireless Information Devices and the Mobile Internet" (presentation)

Egbert-Jan Sol, Ericsson Telecommunicatie BV

- "Making the Internet Mobile for Everybody"

M. Graeber Jordan, GJordan.com LLC

- "The Spirit of the WWW in the Corporate Intranet"

Paul de Bot, Philips Digital Networks

- "The Convergence of Internet and Television Technologies: How it Change the Living Room Experience?"
-

The Ninth International World Wide Web Conference (WWW9) was held on May 15-19, 2000 in the capital of The Netherlands. Leaders from industry, academia, and government presented the latest Web technology, and discussed the issues and challenges facing the Web community as it moves in the 21st Century.

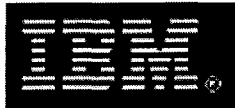
The Centre for Mathematics and Computer Science (CWI) hosted WWW9 and played a key role in the technical program.

The International World Wide Web Conferences are events of the International World Wide Web Committee (IW3C2), and are managed by Foretec Seminars, Inc.

Organizers



Platinum Sponsors



PRICEWATERHOUSECOOPERS

UPC



Gold Sponsors



PHILIPS

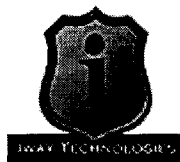
alta^{vista}



Muzmo Communication Inc.



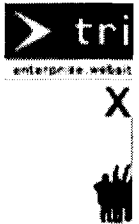
Silver Sponsors



Bronze Sponsors



eXcelon corp.
Dynamic B2B Solutions



Other Sponsors



Google

Related Events of Interest

- 8th International Python Conference
- 16th Unicode Conference
- 2nd Annual Conference on World-Wide Web Applications
- 3rd International Browserday Competition
- HPCN Europe 2000
- Information Outlook 2000
- ACM Hypertext 2000
- ACM Digital Libraries 2000
- ECDL2000

Privacy Statement
Trademark Information

IV. Last Updated: November 29, 2000
webmaster@www9.org

<http://www9.org/w9cdrom/154/154.html>

A. Millau: an encoding format for efficient representation and exchange of XML over the Web

(a) Marc Girardot
Institut Eurécom
Sophia Antipolis, France

girardot@eurecom.fr

Neel Sundaresan
IBM Almaden Research Center
San Jose, California, USA
neel@almaden.ibm.com

2. Abstract

XML is poised to take the World Wide Web to the next level of innovation. XML data, large or small, with or without associated schema, will be exchanged between increasing number of applications running on diverse devices. Efficient storage and transportation of such data is an important issue. We have designed a system called *Millau* for efficient encoding and streaming of XML structures. In this paper we describe the *Millau* algorithms for compression of XML structures and data. *Millau* compression algorithms, in addition to separating structure and text for compression, take advantage of the associated schema (if available) in compressing the structure. *Millau* also defines a programming model corresponding to XML DOM and SAX for XML APIs for *Millau* streams of XML documents. Our experiments have shown significant performance gains of our algorithms and APIs. We describe some of these results in this paper. We also describe some applications of XML-based remote procedure calls and client-server applications based on *Millau* that take advantage of the compression and streaming technology defined by the system.

a) **Keywords:** Binary XML, compression,
tokenization, streaming, proxy server, RPC

3. 1. Introduction

As the World Wide Web transitions from just being a medium for browsing to a medium for commerce, XML (eXtensible Markup Language) [1] has emerged as the standard language for markup. Business to business applications over the Internet are increasingly adopting XML as the *de facto* standard for expressing messages, schema, and data. Consequently, XML is being increasingly used for Web based applications as an exchange wire format. On the other hand, with the popularity of the World Wide Web and increasing dependency on it to find information and to conduct business, the network bandwidth is being tested to its limit. One approach to address this bandwidth problem is to compress data on the network. Traditional data compression algorithms (e.g. Huffman coding [19] or LZ77 [18]) can achieve good compression rates on large text files but are less effective towards small sized files like the ones that may be typical in many *eBusiness* applications. Moreover, they cannot always treat data as a continuous stream. To be really efficient, they need to work on the entire file of a Web object. This is incompatible with the real time constraints of the Web. Finally, these compression systems do not retain this structural information in the data they exchange. Thus there is a need for compression and streaming system that works in the internet context with structured data. These requirements motivate our design for *Millau*.

The Wireless Application Protocol (WAP) [10] defines a format to reduce the transmission size of XML documents with no loss of functionality or semantic information. For example, it preserves the element structure of XML, allowing a browser to skip unknown elements or attributes. *Millau* extends this format to adapt it to business to business applications while improving on the compression algorithm itself. It separates structure compression from text compression. Further, it takes advantage of the schema and data types to enable better compression. To be compliant with the XML standards, it defines a parsing model based on both DOM and SAX at the same time taking advantage of the compression of the document.

This paper is organized as follows. In section 2, we present the work related to compression and more precisely to XML compression. In section 3 we describe the *Millau* compression algorithm. We discuss the DOM and SAX support in *Millau* in section 4. In section 5 we study performance of the *Millau* system and discuss experimental results. In section 6 we briefly discuss some applications of our system. We describe a compression/decompression proxy server system for efficient data exchange and an XML RPC engine which takes advantage of the *Millau* compression model. In section 7 we draw conclusions on our work so far and describe work in progress and future research.

4. 2. Related Work

A lot of work has already been done on lossless data compression [17]. Researchers have developed fast and powerful algorithms for data compression. Their principles are mostly based on Claude Shannon's Information Theory. A consequence of this theory is that a symbol that has a high probability has a low information content and will need fewer bits to encode. In order to compress data well, you need to select models that predict symbols with high probabilities. Huffman coding [19] achieves the minimum amount of redundancy possible in a fixed set of variable-length codes. It provides the best approximation for coding symbols when using fixed-width codes. Huffman coding uses a statistical modeling because it reads and encodes a single symbol at a time using the probability of that character's appearance. A dictionary-based compression scheme uses a different concept. It reads in input data and looks for groups of symbols that appear in a dictionary. If a string match is found, a pointer or index into the dictionary can be output instead of the code for the symbol. The longer the match, the better the compression ratio. In LZ77 compression [18], for example, the dictionary consists of all the strings in a window into the previously read input stream. The deflate algorithm [6] uses a combination of the LZ77 compression and the Huffman coding. It is used in popular compression programs like GZIP [7] or ZLIB [5].

One drawback of these text compression algorithms is that they perform compression at the character level. If the algorithm is adaptive (as, for example, with LZ77), the algorithm slowly learns correlations between adjacent pairs of characters, then triples, quadruples and so on. The algorithm rarely has a chance to take advantage of longer range correlations before either the end of input is reached or the tables maintained by the algorithms are filled to capacity, specially with small files. To address this problem, R. Nigel Horspool and Gordon V. Cormack explore the use of words as basic units of the algorithm [21]. In most implementations of dictionary-based compression, the encoder operates online, incrementally inferring its dictionary of available phrases from

previous parts of the message. An alternative approach proposed by N. Jasper Larsson and Alistair Moffat [22] is to infer a complete dictionary offline to optimize the choice of phrases so as to maximize compression performance.

The Wireless Application Protocol Forum [10] has proposed an encoding format for XML based on a table (the *code space*) that matches tokens to XML tags and attribute names [4]. It takes advantage both of the offline approach (the *code space* can be built offline) and of the word-based compression (tags and attribute names are usually the most frequent words in an XML document). Moreover, unlike the previous compression algorithms, it retains the structure of XML documents. But it does not compress at all the character data content nor the attribute values which are not defined in the DTD. Moreover, it does not suggest any strategy to build the code space in an efficient way. The *Millau* encoding format addresses both of these drawbacks: it is designed to compress character data and defines a strategy to build code space.

5. 3. The *Millau* Compression Model

The *Millau* encoding format is an extension of the WAP Binary XML format. The WBXML (Wireless Application Protocol Binary XML) Content Format Specification [4] defines a compact binary representation of XML. This format is designed to reduce the transmission size of XML documents with no loss of functionality or semantic information. For example, WBXML preserves the element structure of XML, allowing a browser to skip unknown elements or attributes. More specifically, the WBXML content encodes the tag names and the attributes names and values with tokens (a token is a single byte).

In WBXML format, tokens are split into a set of overlapping "code spaces". The meaning of a particular token is dependent on the context in which it is used. There are two classifications of tokens: global tokens and application tokens. Global tokens are assigned a fixed set of codes in all contexts and are unambiguous in all situations. Global codes are used to encode inline data (e.g., strings, entities, opaque data, etc.) and to encode a variety of miscellaneous control functions. Application tokens have a context-dependent meaning and are split into two overlapping "code spaces", the "tag code space" and the "attribute code space":

- The tag code space represents specific tag names. Each tag token is a single-byte code and represents a specific tag name. Each code space is further split into a series of 256 code spaces. Code pages allow for future expansion of the well-known codes. A single token (SWITCH_PAGE) switches between the code pages.
- The attribute code space is split into two numeric ranges representing attribute prefixes and attribute values respectively. The *Attribute Start* token (with a value less than 128) indicates the start of an attribute and may optionally specify the beginning of the attribute value. The *Attribute Value* token (with a value of 128 or greater) represents a well-known string present in an attribute value. Unknown attribute values are encoded with string, entity or extension codes. All tokenised attributes must begin with a single attribute start token and may be followed by zero or more attribute value, string, entity or extension tokens. An attribute start token, a LITERAL token or the END token indicates the end of an attribute value.

In *Millau* format, an *Attribute Start* token is followed by a single *Attribute Value* token, string, entity or extension token. So there is no need to split the attribute token numeric range into two ranges (less than 128 and 128 or greater) because each time the parser

encounters and *Attribute Start* token followed by a non-reserved token, it knows that this non-reserved token is an *Attribute Value* token and that it can be followed only by and END token or another *Attribute Start* token. Thus instead two overlapping code spaces, we have three overlapping code spaces:

- the tag code space as defined in the WAP Specification,
- the attribute start code space where each page contains 256 tokens,
- the attribute value code space where each page contains 256 tokens.

Notice that, in WBXML format, character data is not compressed. It is transmitted as strings inline, or as a reference in a string table which is transmitted at the beginning of the document. In *Millau* encoding format, character data can be transmitted on a separate stream. This allows to separate the content from the structure so that a browser can separately download the structure and the content or just a part of each. This further allows to compress the character data using traditional compression algorithms like deflate [6]. In the structure stream, character data is indicated by a special global token (STR or STR_ZIP) which indicates to the *Millau* parser (see [section 4](#)) that it must switch from the structure stream to the content stream if the user is interested in content and whether the content is compressed (STR) or uncompressed (STR_ZIP). Optionally, the length of the content is encoded as an integer in the structure stream right after the global token (STR_L or STR_ZIP_L). If the length is not indicated, the strings contained in the structure must terminate with a End Of String character or a null character.

We described how *Millau* encoding format efficiently represent character data but we must also take in consideration the fact that, in typical business to business communications, most of the attribute values are of primitive type like boolean, byte, integer or float. For example, in a set of typical business to business XML messages provided by the Open Application Group [14], 70% of the attribute values are of primitive type. These attribute values should not be transcoded in strings in a binary representation of an XML document. So in *Millau*, we use the extension codes to prefix primitives types like bytes, integers or floats. The following table reminds the meanings given to the global tokens by the WBXML Encoding Specification and also precises the meanings of the extension tokens which have been redefined for the needs of *Millau* (these tokens appear in bold in [table 1](#)).

Token Name	Token	Description
SWITCH_PAGE	0	Change the code page for the current token state. Followed by a single u_int8 indicating the new code page number.
END	1	Indicates the end of an attribute list or the end of an element.
ENTITY	2	A character entity. Followed by an integer encoding the character entity number.
STR_I	3	Inline string. Followed by a string.
LITERAL	4	An unknown tag or attribute name. Followed by an integer that encodes an offset into the string table.

FALSE	40	Encodes the boolean value false.
TRUE	41	Encodes the boolean value true.
FLOAT	42	Inline float. Token is followed by an integer representing the floating-point argument according to the IEEE 754 floating-point "single precision" bit layout.
PI	43	Processing instruction.
LITERAL_C	44	Unknown tag, with content.
STR_L	80	Indicates that uncompressed character data has been written to the content stream. Followed by an integer indicating the number of characters.
STR_ZIP_L	81	Indicates that compressed character data has been written to the content stream. Followed by an integer indicating the number of characters.
EXT_T_2	82	Inline integer. Token is followed by an integer.
STR_T	83	String table reference. Followed by an integer encoding a byte offset from the beginning of the string table.
LITERAL_A	84	Unknown tag, with attributes.
STR	C0	Indicates that uncompressed character data has been written to the content stream.
STR_ZIP	C1	Indicates that compressed character data has been written to the content stream.
BYTE	C2	Inline byte. Followed by a single byte.
BINARY	C3	Binary data. Followed by an integer indicating the number of bytes of binary data.
LITERAL_AC	C4	Unknown tag, with content and attributes.

(a) *Table 1: Global WBXML and Millau tokens*

The following is an example of a simple tokenized XML document. Here is the source document:

```
<?xml version="1.0" encoding=?UTF-8??>
<!DOCTYPE Book [
  <!ELEMENT Book (Title, Chapter+, Picture+)>
  <!ATTLIST Book
    Author CDATA #REQUIRED
    Genre (literature|science|history|cartoons) #REQUIRED >
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Chapter (#PCDATA)>
  <!ATTLIST Chapter
    id ID #REQUIRED>
```

```

<!ELEMENT Picture (#PCDATA)>
<!--ATTLIST Picture
Caption CDATA #REQUIRED-->
]>
<Book Author="Anonymous" Genre=?literature?>
  <Title>Sample Book</Title>
  <Chapter Number="1">
    This is chapter 1. It is not very long or interesting.
  </Chapter>
  <Chapter Number="2">
    This is chapter 2. Although it is longer than chapter 1, it is not any more interesting.
  </Chapter>
  <Picture Caption="Nice picture">
    [base 64 encoded binary data]
  </Picture>
</Book>

```

Tokens for the tag code space, the attribute names code space, and the attribute value code space are defined in [table 2](#).

Tag code space		Attribute name code space		Attribute value code space	
Tag Name	Token	Attribute Name	Token	Attribute Value	Token
Book	5	Author	5	literature	5
Title	6	Genre	6	science	6
Chapter	7	Number	7	history	7
Picture	8	Caption	8	cartoons	8

(b) **Table 2: Code space example**

Tokenized form (numbers in hexadecimal) follows:

```

01 01 6A 00 C5 05 03 "Anonymous" 06 05 01 46 C1 C7 07 C2 01 01 C1 01 C7 07 C2
02 01
C1 01 C8 08 03 "Nice picture" 01 C3 ... 01 01

```

6. 4. *Millau* API - Specification and Implementation

The *Millau* format is designed to represent XML documents in a compact way using tokens to represent tags and attributes instead of strings. We built parsers for documents encoded using this format implementing the two standard APIs DOM [3] and SAX [2]. DOM is the tree model API used to represent and process parsed XML document trees. The SAX API has an event-based streaming model typically used to process large XML documents without actually building a parse tree.

We provide two variants of SAX parsers. The first one produces traditional SAX events, as defined by the SAX API. This means that each time it encounters a tag token, it generates a *startElement* event passing the name of the tag. The name matching the tag token is found in the code spaces. We will describe later how the code spaces are built in our implementation. The second SAX parser, which we call the *Millau* BSAX (Binary SAX) parser, extends the SAX API by providing events which pass tokens instead of strings. This parser has been designed for applications that are able to handle tokens instead of strings. We show later how applications using tokens perform better than those using strings only.

We also provide two DOM-based parsers. The first one creates a conventional DOM tree from a *Millau* stream. The second one creates what we call a BDOM tree (Binary DOM tree). A BDOM tree is like a DOM tree but instead of storing node names it stores, for each node, a pair (*page number*, *token*) which uniquely identifies the node. Here follows a description of how each parser works.

a) 4.1 The *Millau* SAX parser

A conventional SAX parser parses an XML stream and throws SAX events (e.g. *characters*, *startElement*, *endElement*) that can be handled by a specific handler. Parameters can be passed through these events (e.g. the element name is passed through the *startElement* and *endElement* events). These events and their associated parameters are defined by the SAX API [2]. The *Millau* SAX parser has been designed to parse a *Millau* stream. It implements the SAX API. In the following paragraphs, we describe how it works.

Before reading tokens from the binary input stream, the *Millau* SAX parser creates a LIFO (last in - first out) stack in which it puts the names of the element that are opened and not yet closed. This is so that it can get the name of an element when it ends and send it to the handler. Then it reads tokens from the input stream until the stack is empty. When the stack is empty, it means that the root element has been closed. Table 3 specifies the action taken for each token type.

Token	Action Taken
switch page	read the next token which gives the current code page.
string inline	read the inline string that follows and throw a character event.
extension	read the following content according to its type, translates it into a string and throw a character event.
end token	remove the last element of the tag names stack and throw an <i>endElement</i> event with the tag name which has been removed from the stack.
not a reserved token	if the token is not a reserved token, then it is a tag token, so the parser looks for the corresponding tag name in the element code space (if not found, an exception is raised). It then calls a method which returns an attributes list. Eventually, it throws a <i>startElement</i> event with the tag name and its corresponding attribute list (if the element has attributes)

(a) **Table 3: Millau element tokens decision table**

The *getAttribute* method tests the most significant bit of the tag token to know if this element has attribute. If the bit is 0, the element has no attribute and the method returns an empty list. If the bit is 1, the element has attributes and the method reads the attribute tokens from the input stream.

While the most significant bit of the next read tokens is 0, the parser knows that these tokens are not attribute value token. The tokens are processed, based upon their types, as described in table 4.

Token	Action Taken
switch page	read the next token which gives the current code page
not a reserved token	if it is not a reserved token, then it is an attribute name token. So the parser looks for the corresponding name in the attribute name code space (if not found, an exception is raised). It then reads the attribute value.
end token	end of the attribute list identified; return the attributes list.

(b) **Table 4: Millau attribute tokens decision table**

The attribute value can be encoded as a token value, as an inline string (compressed or not) or as a primitive type like byte, integer, float, or boolean.

b) 4.2 The *Millau* Binary SAX Parser

It is expected that parsing a compressed *Millau* stream using our SAX parser is faster than decompressing a compressed XML stream and then parsing it with a conventional SAX parser. But it could take more time than parsing a non compressed XML stream with a conventional SAX parser. We observed that the part of the processing which takes the most time with *Millau* SAX parser is the translation of the tokens in elements and attributes names. The reason for this is that, for each received token, the parser must search the code spaces for the corresponding strings. For example, if it receives an element token, it must search the corresponding element name in the element code space and this can take a lot of time, especially if there are many elements in the element code space. Skipping this translation step could make the encoded XML parsing faster. These tokens do not really need to be translated into strings at all. In fact, they can be directly processed by appropriate handlers which recognize the tokens. The design of such handlers and the efficiency aspects will be discussed later. A *Millau* Binary SAX parser is like a SAX parser but instead of studying character based XML streams it operates on the binary encoded XML. Instead of passing tag names and attribute names and values to the handler, it passes encoding tokens without translating them into strings. More precisely, each time it throws a *startElement* event or an *endElement* event, it passes a pair (*code page*, *element token*) which uniquely identifies

the element (See WBXML encoding specification [4] or [section 2](#)). For a *startElement* event, it also passes a *Millau* binary attribute list which is a variant of the XML SAX attribute list implementation. A *Millau* attribute list, instead of containing triples (*attribute name*, *attribute type*, *attribute value*) contains triples (*attribute name uid*, *attribute type*, *attribute value uid*) if the type of the attribute is "enumerated" or triples (*attribute name uid*, *attribute type*, *attribute value*) if the type of the attribute is "CDATA". A "uid" (unique identifier) is a pair (*code page*, *token*). It can uniquely identify an attribute name or an attribute value. [Table 5](#) illustrates the differences between the two parsers:

Interface	<i>Millau</i> conventional SAX parser	<i>Millau</i> Binary SAX parser
Handler	startElement(String name, AttributeList)	startElement(int token, BAttributeList)
AttributeList	getName(int i) returns the name	getNameToken(int i) returns a token
AttributeList	getValue(int i) returns the value	getValueToken(int i) returns a token getValue(int i) returns an Object

(a) **Table 5: Comparison between *Millau* conventional and binary SAX parsers**

A *Millau* SAX handler must be able to recognize (*code page*, *token*) to trigger special processing adapted to the element or the attribute. It is faster than a conventional handler because, instead of comparing two strings (a time consuming operation), it just has to compare two pairs of bytes.

c) **4.3 The *Millau* DOM parser**

The DOM parser is able to build a DOM tree dynamically from a binary XML stream. The top-level architecture of the DOM parser is almost the same as the architecture of the SAX parser. Like the SAX parser, the DOM parser creates a LIFO stack to store the names of the opened elements. Then it reads tokens from the input stream until the stack is empty. It differs from the SAX parser in the processing which is done for each type of token. [Table 6](#) gives the details:

Token	Action Taken
switch page	read the next token which gives the current code page.
string inline	reads the inline string that follows and creates a text node and appends this text node to the last opened element (the first element of the LIFO stack).
extension	read the content following the content according to its type, translates it into a string, creates a text node and appends it to the last opened element.
end token	just remove the last element of the tag names stack.
not a reserved	if this is not a reserved token, then it is a tag token, so looks for the corresponding tag name in the element code space (if not found, an

token	exception is raised). It then creates an element node. If the stack is empty, it means that this element is the root of the document, so it is appended to the document node. If the stack is not empty, the element is appended to the last opened element (the first in the LIFO stack). Eventually, the parser tests the last bit of the token, if it is 1, it invokes a method which gets the attributes for this element.
-------	--

(a) **Table 6: Millau DOM parser decision table for element tokens**

The *getAttributes* method reads the attribute tokens from the binary XML stream. While the most significant bit of the next read tokens is 0, the parser knows that these tokens are not attribute value tokens. Table 7 describes the action taken on different kinds of tokens:

Token	Action Taken
switch page	it reads the next token which gives the current code page.
end token	this is the end of the attribute list. The method exits.
not a reserved token	if it is not a reserved token, then it is an attribute name token. So the parser looks for the corresponding name in the attribute name code space (if not found, an exception is raised). It then reads the attribute value. Eventually, it adds this attribute to the current element.

(b) **Table 7: Millau DOM parser decision table for attribute tokens**

The attribute value can be encoded as a token value or as an inline string (compressed or not).

d) 4.4 The Millau Binary DOM parser

The *Millau* Binary DOM parser uses the Binary DOM (BDOM) API. Table 8 describes the action taken by the parser on different types of tokens:

Token	Action Taken
switch page	no change
string inline	no change
extension	the BDOM parser can create a primitive type node (boolean, byte, integer, float, binary data) defined by the BDOM API by invoking the methods <i>createBooleanNode</i> , <i>createByteNode</i> , <i>createIntegerNode</i> , <i>createFloatNode</i> or <i>createBinaryData</i> of the class <i>BDocument</i> . This node is then appended to the last opened element by invocation of the method <i>appendChild</i> .
end token	no change

tag token	the BDOM parser creates a <i>BElement</i> node by invoking the method <i>createElement</i> of the class <i>BDocument</i> with a short as parameter. The first byte (most significant) of this short is the code page of the tag and the second byte is the tag token.
-----------	---

(a) **Table 8: Millau Binary DOM parser decision table**

e) 4.5 The Millau Binary DOM API

The Binary DOM API implements all the interfaces of the DOM API as defined by the W3C [3]. The main advantage of a Binary DOM tree is that the tag and attribute names are stored not as strings but as tokens and is space-efficient. The correspondence between names and pairs (*code page*, *token*) is stored in the code spaces so that names can be normally retrieved for every element or attribute nodes. Attribute values can be stored as tokens, if available, as strings, or as primitive types. The primitive types currently supported by the BDOM API are boolean, byte, integer (4 bytes), and float. Element contents can also be stored as primitive types. For element contents, we have defined one more Binary Node, the Binary Data Node, which stores binary data without base 64 encoding, thus avoiding the 33% overload of the base 64 encoding. This is useful for binary files like images embedded in an XML document.

In addition to the methods of the DOM API, the BDOM API also provides methods for creating or retrieving elements or attributes by tokens instead of strings. This is useful for applications which have been designed to work with *Millau* format (see [section 6](#)). For example the class *BElement* (for Binary Element) which implements the DOM interface *Element* has also a method *getTagToken()* which returns a short where the first byte is the code page and the second page is the tag token. For convenience, the class *BDocument* which implements the DOM interface *Document* provides a method *writeBinaryXML(OutputStream)* which write the BDOM tree in *Millau* format to the *OutputStream*.

f) 4.6 The Millau Code Spaces

The choice of the data structure to represent the code spaces is also important for good performance of the system. The translation time is mostly influenced by the time it takes to look up in the code spaces for a token or for its corresponding string. Depending of what the program needs to do, translating strings into tokens or tokens into strings, different data structure may be used. For example, to convert strings into tokens quickly, strings must be found quickly in a table. For this, it is better to use a hash table where the keys are the strings and the values are the corresponding tokens. But, if given a page number and an index in a code page the corresponding string must be found quickly, the best data structure is a two dimensional array indexed by page numbers and indexes in pages. If we need to be able to find a string from a token quickly or a token from a string, then we need to sort the table and then do a binary search to find a string corresponding to a token.

Next we describe the method to fill in the hash table for element code space. First, the page number variable is set to 0 and the index variable to 5 (the first four indexes are

reserved for global tokens). For each element declaration, the system gets the element name, adds it in the hash table with the element name as the key and $(56 \times \text{pageNumber} + \text{index})$ as the value. The system increments the index by 1. The size of a page for elements is 64 because the last two bits of the index are reserved so when the index reaches the value 64, the system increments the page number by 1 and resets the index to 5. When the page number reaches its maximum value 255, an exception is raised.

For each element declared, the system gets the corresponding attribute declaration from the previously built DOM tree. It adds the attribute name in the hash table with the attribute name as the key and $(256 \times \text{pageNumber} + \text{index})$ as the value. If the attribute type is enumerated (enumerated attribute types are NOTATION or NAME_TOKEN_GROUP), then the system looks for the values of this enumerated attribute. For each value, it adds the attribute value in the hash table with the attribute value as the key and $(256 \times \text{pageNumber} + \text{index})$ as the value. The system increments the index for the value by 1. The size of a page for attribute value is 128 so when the index reaches the value 128, the system increments the page number by 1 and resets the index to 5. When the page number reaches its maximum value 255, an exception is raised. If there are no values or when the values have been successfully added to the attribute value code space, the system increments the index for the name by 1. The size of a page for attribute name is 128 so when the index reaches the value 128, the system increments the page number by 1 and resets the index to 5. When the page number reaches its maximum value 255, an exception is raised.

Next, the method to fill in the 2-dimensional array for element code space is described. First, we set the page number variable to 0 and the index variable to 5 (the first four indexes are reserved for global tokens). For each element declaration, the system gets the element name, adds it in the elements array at position (page number, index). The system increments the index by 1. The size of a page for elements is 64 because the last two bits of the index are reserved so when the index reaches the value 64, the system increments the page number by 1 and resets the index to 5. When the page number reaches its maximum value 255, an exception is raised.

The attribute names code space and the attribute values code space can be merged into one so that each pair (*attribute name*, *attribute value*) is a single token instead of two tokens (name and value). The code space is filled as follows. For each element declared, the system gets the corresponding attribute declaration from the previously built DOM structure. If the attribute type is not enumerated (no specific value is declared for this attribute), then the system adds the attribute name in the attribute code space (hash table for the server, array for the client). If the attribute type is enumerated, then the system looks for the values of this enumerated attribute. For each value, it adds the pair (*attribute name*, *attribute value*) with a specific token in the attribute code space. When the server comes across an attribute with a value, it looks in the attribute code space for the couple (*attribute name*, *attribute value*). If it can find it, it sends this token. If it cannot find it, it looks for the attribute name in the attribute code space. If the name is found, the server sends the corresponding token for this name followed by a string inline token followed by the attribute value encoded in the charset specified at the beginning of the binary XML stream. If the name is not found, an exception is raised.

Attributes may be mandatory (*#REQUIRED*), optional (*#IMPLIED*), or can have fixed values (*#FIXED*). For mandatory or fixed attributes, it is not necessary to transmit tokens. To achieve this optimization the system can store in the element code space the names of the required or fixed attributes with the element name. For example, if attributes *Author* and *Genre* are required for element *Book*, the element code space stores the triplet (*Book, Author, Genre*) at the entry *Book*. This element code space is filled as follows. For each element declaration, the system gets the element name and the required and fixed attributes. It adds the element names and the required and fixed attribute names to the element code space. For the fixed attributes, it also adds their value. In the attribute code space, only the implied attributes will be stored with their corresponding values (if defined).

Notice that for applications which can work with tokens without translating them into strings, there is no need for code spaces. This saves memory and CPU. However, to facilitate the task of the developer of the application, the tokens can be stored as static variables with explicit names.

7. 5. Experimental results

In this section we study the performance of the *Millau* system.

a) 5.1 Theoretical compression rate

First we compute the theoretical size of *Millau* streams. Suppose:

- there is a total of N occurrences of elements of which M have empty contents and K are elements with attributes,
- there are a total of T text elements,
- there are a total of A attribute occurrences,
- S is the total size of the text elements in bytes,
- H is the size of the header and,
- P is the number of page switches.

The size W of the token encoded document can be computed as: $W = 2N - M + 2A + K + T + S + 2P + H$

The following explains the origin of each term in the above formula:

- $2N$: each element is represented by an element token plus and end token (2 bytes)
- M : empty elements do not need an end token
- $2A$: each attribute is represented by a name token and either an attribute token, a string inline token plus a string or an extension token plus a primitive type (at least 2 bytes per attribute)
- K : every attribute list ends with an end token
- T : each text element is introduced by a string inline token
- S : size of all the text which has no corresponding token (it can be text elements or unknown attribute values)
- $2P$: each page switch means one switch token plus one byte for the new page number
- H : the header is composed of a version number (one byte), a document public identifier (1 to 4 bytes or one byte plus a string), a character set (1 to 4 bytes), a string table (1 to 4 bytes plus possibly the strings)

The first five parameters N , M , A , K , and T depend only on the document so we cannot change these parameters for a given document. The size of the text can be a large

parameter. First of all, the number of unknown attribute values must be minimized. This can be done by encoding the attribute values given by the DTD for the attribute of the type enumerated. Another improvement is to pre-parse the document and look for the attribute values which are not in the DTD and appear more than once in the document. These attribute values can be put in the string table so that they will appear only one time in the binary XML stream.

The best solution to text compression is to separate the text from the document structure and compress it with an algorithm like GZIP [7]. The same can also be done for the unknown attribute values but it can cost in terms of processing time to have to switch to another stream for each attribute value. The size of the header depends mainly of the use of a string table : a string table contains the most frequently used strings in the document. They are then referenced in the document by an offset in the string table. The experiments show that the number of page switches can constitute a large overhead (a page switch is two bytes). This number can be further reduced by putting sibling elements on the same page. The DTD can be used to perform this optimization. (The number of page switches cannot be larger than $2(N + 2A)$ because there cannot be more page switches than the number of elements and attributes names and values.)

The following formula now gives a good estimate of the size X of the XML stream: $X = (2N - M)n + A(a + v) + S$ where n is the mean length of an element name, a is the mean length of an attribute name, and v is the mean length of an attribute value.

The theoretical estimated compression rate is thus given by this formula :

$$C = W / X = (2N - M + 2A + K + T + S + 2P + H) / ((2N - M)n + A(a + v) + S)$$

We can compute a better estimate by using the exact size X of the XML stream if it is known. Usually, it is not an unknown parameter. We will use this value in the experimental computations. We observe that C is a function of the size of the markup. If elements and attributes are given long and explicit names in the original XML document, the compression rate can be very good.

b) 5.2 Experimental compression rate

To measure the performance of our compression system, we have built a test package which considers the following parameters:

- number of elements
- number of empty elements
- number of attributes
- number of elements with attributes
- number of text elements (a.k.a. PCDATA elements)
- number of page switches
- compression rate for the structure stream encoded in *Millau* format
- compression rate for the data stream compressed with GZIP algorithm
- compression rate for the whole document processed by our compression system
- compression rate for the whole document compressed with GZIP
- compression rate for the structure stream compressed with GZIP
- time to parse an uncompressed XML stream
- time to parse an XML streamed compressed with GZIP
- time to parse a *Millau* stream
- time to parse a *Millau* stream without decoding the name and value tokens

We ran our experiments with XML documents of varying sizes. We first discuss a typical set of results obtained with a 3 MB technical manual marked up in XML. This document is a valid XML document which means that it comes with a DTD which can be used to efficiently build the code spaces. Table 9 presents the results:

Size of the document (in bytes)	3 093 194
Size of the markup (in bytes)	2 038 952
Size of the character data (in bytes)	1 052 242
Number of elements	73 591
Number of empty elements	9 485
Number of attributes	25 611
Number of elements with attributes	22 529
Number of PCDATA elements	64 588

(a) **Table 9: Size of the markup for a technical documentation**

From the above table, we can compute a lower bound for the theoretical compression rate for the markup (PCDATA excluded) as defined previously. This is a lower bound and not an exact value because the number of page switches is not known a priori. Moreover, we assume that there are no attribute values encoded as strings (which is usually not true). In other words, we assume that every attribute value has a corresponding token. The lower bound of the size of the markup encoded in *Millau* format is thus given by: $2 * 73591 - 9485 + 2 * 25611 + 22529 + 64588 = 276036$ bytes. Subsequently, the lower bound of the theoretical compression rate for the markup is given by: $276036 / 2038952 = 13.5 \%$.

Notice that the number of page switches cannot be larger than : $2 * (73591 + 2 * 25611) = 249626$ bytes. So the upper bound for the *Millau* size is (if there are no attribute values encoded as strings): $276036 + 249626 = 525662$ bytes. Subsequently, the higher bound of the theoretical compression rate for the markup is: $525662 / 2038952 = 25.7 \%$. This seems reasonably good for a worst case measure.

Table 10 shows the experimental compression rate achieved by *Millau* algorithm and GZIP algorithm (sizes are given in bytes):

	Initial document size	<i>Millau</i> document size	<i>Millau</i> compression	GZIP document size	GZIP compression
Whole document	3 093 194	593 795	19.2%	401 518	13.0%
Markup only	2 038 952	367 321	18.0%	175 044	8.6%
Data only	1 052 242	226 474	21.5%	226 474	21.5%

(b) **Table 10: Comparison of the compression rates achieved by Millau and GZIP algorithms**

The figures compared are the compression rate of the markup using *Millau* encoding and using GZIP encoding (figures in bold in the chart). It can be seen that the compression rate achieved by *Millau* encoding is reasonable (18.0% is usually considered as a good compression rate for text). But we can also notice that the compression rate achieved by the GZIP algorithm for the same markup is very good (8.6%). Actually, this is not very surprising because the GZIP algorithm takes advantage of the redundancy of a document to compress it and the XML markup is highly redundant. There are a few tags which are repeated a large number of times in this large document. This is why the compression rate for GZIP outperforms the compression rate for *Millau*. But at the same time, GZIP does not retain structure of the document which is a disadvantage for fast documents processing.

To improve the compression rate of *Millau*, we can limit this number of page switches (about 30000) by reordering the code spaces so that opening tags which are close to each other in the document appear on the same code space page. We do not care about the closing tags because there are encoded with the reserved token END in *Millau* format. The code space optimization can be done from the DTD or from the XML document itself. To formalize the problem of optimizing from the XML document itself, we consider an ordered set T of n possible tags where n is bigger than the size p of a code space page. The document can be represented as a series of tags (t_i) where each t_i is element of T . We want to find a permutation of T that minimizes the number of page switches inside of this document. The first approach is to find the most frequent pair (t_i, t_j) (a pair is two different tags that appear next to each other in the XML document) and put them in the same page. A better approach consists in computing the mean distance between two tags for each pair (t_i, t_j) and then grouping tags in fixed-size clusters using a clustering algorithm. Each cluster here represents a code space page.

We are currently working on optimizing the code spaces from the DTD. From the DTD, we can estimate the probability that a tag follows another tag. The next step is to compute the probability that two tags will follow each other in less than p hops (where p is a page size). From these probabilities, we can compute mean distances between tags in order to apply a clustering algorithm as we do with the XML document itself. An alternative to the code spaces approach is to encode the tags with variable length tokens. One or several bytes encode a tag according to its occurrence frequency. The 128 most frequent tags will be encoded with a single byte. The formats of these bytes is similar to the byte format of UTF-8 [23]. Table 11 shows the byte sequences:

Token length	Byte sequence
1 byte	0xxxxxxx
2 bytes	110xxxxx 10xxxxxx
3 bytes	1110xxxx 10xxxxxx 10xxxxxx
4 bytes	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

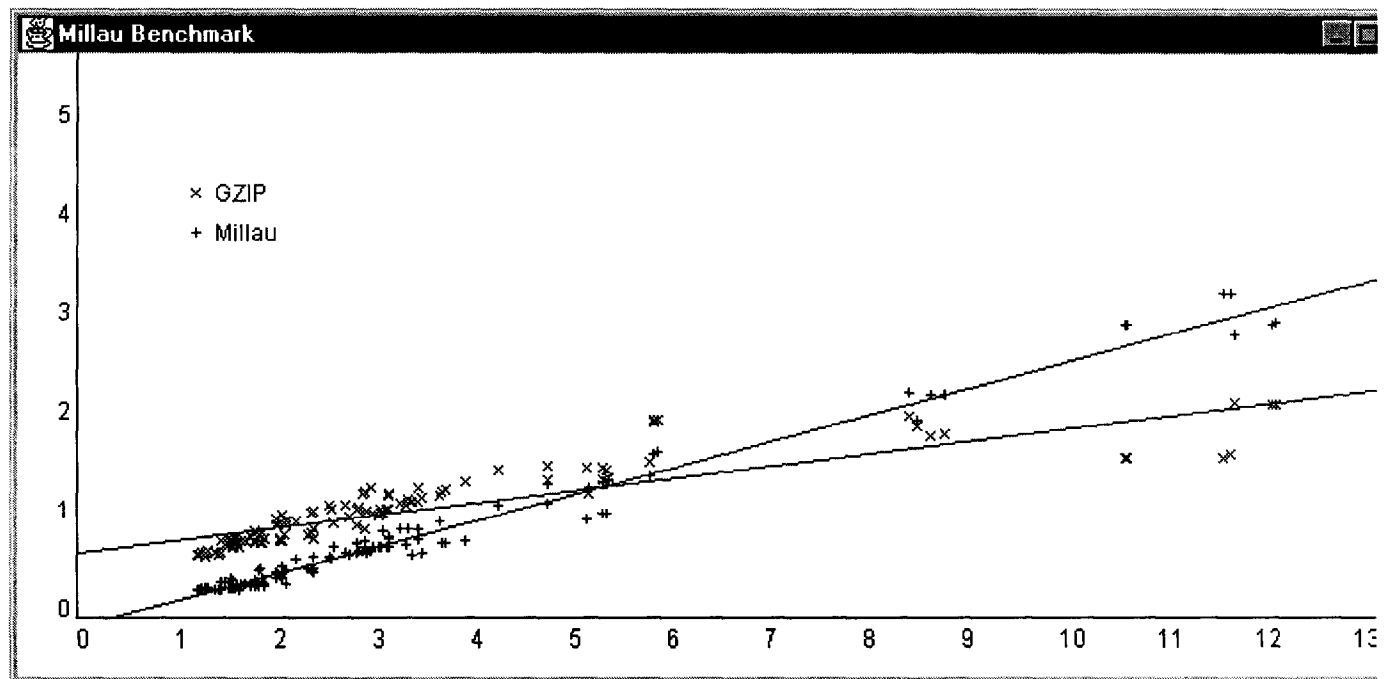
5 bytes	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
6 bytes	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

(c) **Table 11: UTF- 8 byte sequences**

c) 5.3 Influence of the document size

In the previous section we computed the compression rate for a big XML document. This experiment has shown that, for this big document, GZIP was more efficient than *Millau* as regards the compression rate because GZIP takes advantage of the redundancy induced by the XML markup. It has also highlighted some theoretical problems that will need to be solved in further investigation. In this section we run experiments on a set of small XML files like the ones exchanged in business transactions and see if *Millau* can be more efficient with this kind of files than traditional compression algorithms like GZIP.

We have tested *Millau* compression algorithm on a set of 118 XML reference files from the Open Applications Group [14]. These sample files represent the typical files which are used in business transactions (e.g. sales orders, bills, payments). Their sizes range from 1KB to 14KB. They all come with a specific DTD. We compressed them, first using GZIP algorithm [7] with a buffer size of 512 bytes, then using *Millau* on the XML stream. The *Millau* code spaces have been built from the DTDs provided with the XML sample files. On the diagram of figure 1, we represented the size of the compressed files with respect to the initial size. The red points are for GZIP while the blue ones are for *Millau*.



(a) **Figure 1: Size in KB of the compressed file (red: GZIP, blue: Millau) w.r.t. the initial size**

First, we observe that the size of the *Millau* compressed file grows linearly with the file of the uncompressed file (the correlation coefficient between the *Millau* compressed size and the initial size is 0.9918 which is very close to 1). This means that the compression ratio is roughly constant (~ 20%) with respect to the document size. On the other hand, the compression rate of GZIP grows with the size of the initial document. This is because GZIP (like other lossless traditional compression algorithms) is designed to take advantage of the redundancy of character strings inside of a document. That is also why the compression of GZIP does not grow linearly (the correlation coefficient between the GZIP compressed size and the initial size is only 0.8995 to compare to 0.9918). Although it is true that the redundancy is high on large XML documents because of the limited number of allowed tags, it is also found that the occurrence frequency of repeated characters in a small XML document (like a sales order) might not be very high (because of the limited size of the document). So one can think that it is beneficial to map XML tags to tokens and that this mapping must be done offline from the DTD. The experience proves that this is beneficial for small documents (typically documents which size ranges from 0 to 5 KB -according to the crossing point between the two regressions lines --this is the case of most of the documents exchanged in business transactions-- here, only 20 of the 118 documents of our eBusiness test set are bigger than 5 KB).

d) 5.4 Processing times

The third experiment measures the time to parse an XML stream (with and without compression) and compares that time to the time needed to parse a *Millau* stream. For the XML stream, we have used the SAX driver of the IBM XML parser [24]. For the *Millau* stream, we have obtained measurements with two kinds of parsers :

- a *Millau* SAX parser,
- a *Millau* BSAX parser.

As a reminder, the *Millau* SAX parser implements the Parser interface defined by SAX so it must translate every token to its corresponding string. On the other hand, the *Millau* BSAX parser generates events with tokens instead of strings so it does not make any translation (see section 4).

For the measurements, the XML stream is created from a 3MB technical manual marked up in XML and served of a local disk. It is better to use a large file for timing experiments because the experiments have shown that the method *currentTimeMillis* of the Java class *System* was not able to measure time differences smaller than 10 milliseconds. SAX events are handled by basic handlers which do no processing at all. We measured the time to parse the stream from the creation of the stream to the endDocument event. Table 12 summarizes the results:

Time to parse the uncompressed XML stream	40 seconds
Time to parse the XML stream compressed with GZIP	40 seconds
Time to parse the <i>Millau</i> stream using a <i>Millau</i> SAX parser	8 seconds
Time to parse the <i>Millau</i> stream using a <i>Millau</i> BSAX parser	5 seconds

**(a) Table 12: Parsing time comparison
using IBM SAX parser and Millau SAX
parser**

First, this indicates that the difference between parsing a compressed or uncompressed stream is negligible. Secondly, parsing a *Millau* stream is at least five times faster than parsing an ASCII XML stream. This result was initially somewhat surprising. However, this result can be explained by the fact that the operations performed during the parsing of a token stream are easier and less time consuming than the operations needed to parse a text XML stream: the parsing of a text stream involves string comparisons, a time consuming operations while the parsing of binary stream involves bytes comparison. Moreover, the fact that the content stream (which does not need any parsing) is separated from the structure stream makes the parsing more efficient. The processing speed is one of the greatest strengths of *Millau* and is the main reason why this model is particularly well adapted for business applications as well as streaming applications.

We can also observe that the time needed to translate tokens into strings by performing a look-up in the code space (3s to compare with 5s to parse the stream) is not a large overhead. This can be done if the application has not been designed to work with tokens. But for efficiency purposes it is preferable to work with tokens instead of strings as the time needed by the handler to compare two strings can be much larger than the time needed to compare two tokens.

8. 6. Applications using *Millau* Streams

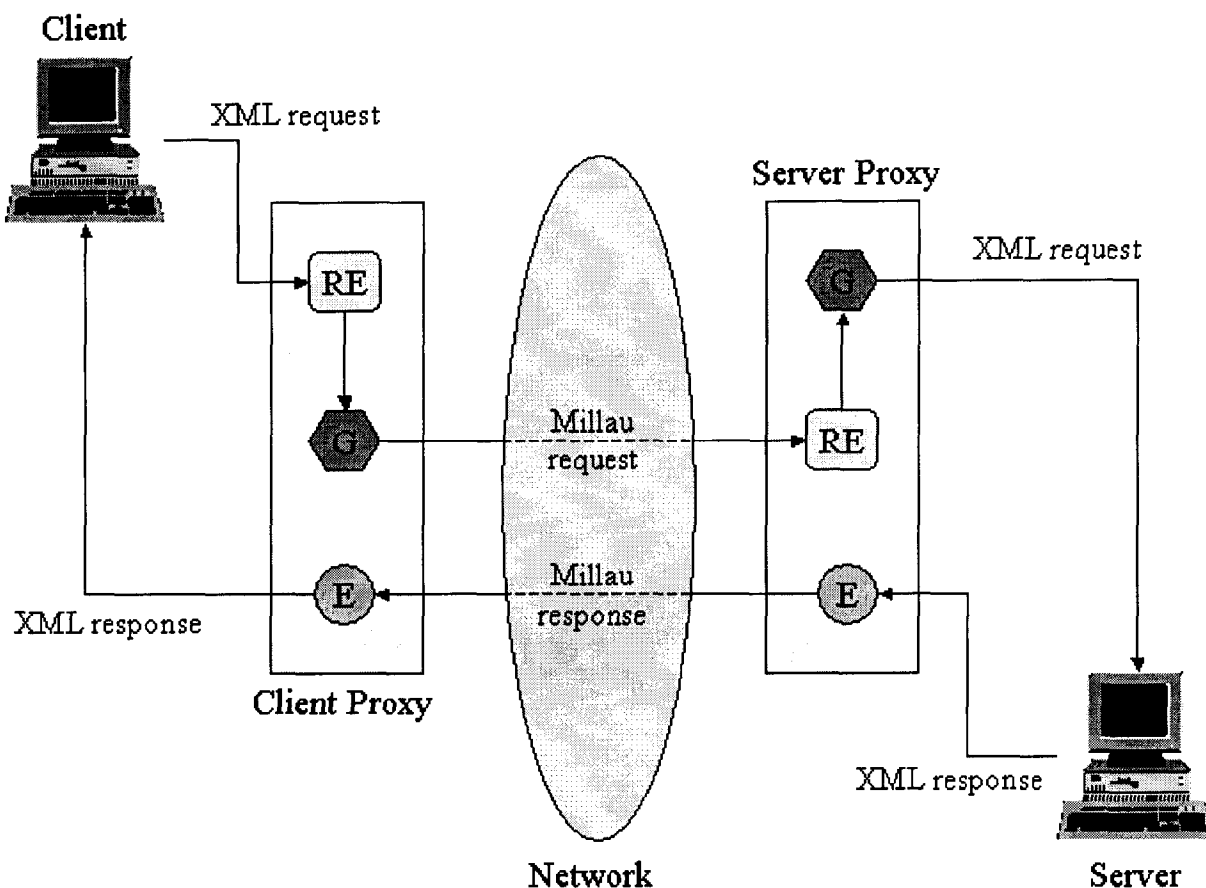
We have built two applications using the *Millau* APIs. The first application is a *Millau* compression Proxy Server and its companion *Millau* decompression Proxy Server. The second application is an implementation of the XML RPC using *Millau* as an exchange data format. For these two applications, we discuss the implementation design and why these are good applications of *Millau* encoding format.

a) 6.1 *Millau* Compression Proxy Server

Studies have been made in prior research work in using HTTP proxy servers to compress data on the network [9] [11] [16]. These demonstrate that about 33% of the bandwidth can be saved easily by compressing the data exchanged on the network. In these studies, they have used conventional compression algorithms like ZLIB and GZIP to compress the text data. The fact that a significant portion of the Web objects electronic commerce have a small size makes these compression algorithms not efficient for this purpose. So it has been suggested that compression proxy server uses a static table that maps the frequently used HTML strings into tokens. However, they do not have a systematic scheme like ours to compress any XML document. Here, we investigate the architecture and the performances of an HTTP proxy server based on *Millau* Binary XML.

The system is composed of two proxy servers: a *client proxy server* and a *server proxy server*. The client proxy server is located "close" to the client, and the server proxy server is located "close" to the server. In an extreme situation the client proxy server

may be merged with the client, and the server proxy server with the server. Also if there is seamless XML data flow between two locations in both directions there may be possibly more than one compression proxy server and decompression proxy server. We built our proxy servers using the WBI (Web Intelligence) proxy architecture system [15]. Before describing the architecture with more details, we introduce the architecture of WBI. WBI is a programmable HTTP request and HTTP response processor. WBI's data model is based on the request/response structure of HTTP version 1.0. Each request and each response consist of a *structured* part and a *stream* part. The structured part corresponds to the header and the stream part corresponds to the body. *Millau* mainly works on the stream part which is here supposed to be XML content. Notice that HTML content can be converted to XML thanks to an already existing WBI plugin so that *Millau* can then process this stream. WBI receives an HTTP request from a client, such as a Web browser, and produces an HTTP response that is returned to the client. The processing that happens in between is controlled by the modules programmed into WBI. A typical WBI transaction flow goes through three basic stages: request editors (RE), generators (G), and editors (E). A request editor receives a request and has the freedom to modify the request before passing it along. A generator receives a request and produces a corresponding response (i.e. a document). An editor receives a response and has the freedom to modify the response before passing it along. When all the steps are completed, the response is sent to the originating client. The diagram of [figure 2](#) shows the architecture of the *Millau* compression-decompression client proxy server and its associated compression-decompression server proxy server.



(a) **Figure 2: Workflow path of a Web request and its response in the Millau compression-decompression WBI proxy environment**

Here, we describe the flow for a request-response transaction between a Client and a Server. The Client generates an XML request and expect an XML response from the Server. In the Client Proxy, the Request Editor (RE) compress the XML body of the HTTP request into a *Millau* stream using a *Millau* XML Tokenizer. The request is then handled by the Generator (G) which forward the request, now in *Millau* format, to the Server Proxy. In the Server Proxy, the *Millau* request is received by the Request Editor which decompresses the *Millau* stream using a *Millau* SAX Parser. The request, now in XML format, is handled by the Generator which forwards the request to the Server. The Server produces a response which is received by the Editor (E) of the Server Proxy. The Editor compresses the response in *Millau* format using an XML Tokenizer and send this encoded response to the originated client (the Client Proxy). The Editor of the Client Proxy decodes the response using a *Millau* SAX Parser and send it to the Client. Notice that the Client Proxy is actually a simple WBI plugin that can be embedded in a LAN proxy cache machine or that can also be running directly on the client machine. An

important issue is that the client proxy is "close" to the client machine. By "close" we mean that the connection between the client machine and the client proxy machine is fast compared to the connection between the client and the server (typically an Internet connection). For example, the client proxy machine must be on the same LAN than the client machine. In the same way, the server proxy must be "close" to the server machine.

To test the efficiency of this architecture, we compare the compression-decompression overhead with the download time saving that *Millau* compression realizes. We measured this compression-decompression overhead on a large set of small and big XML documents. The results show that the compression-decompression overhead is small compared to the download time that can be saved, specially on a low bandwidth connection like a modem connection. In the following table, we present mean results for small eBusiness documents and for large text documents (Shakespeare's plays encoded in XML by John Bosak [8]). The transmission time was computed for a 56Kbits/s modem line. Table 13 shows the results.

Document Type	Mean size	Transmission time	Compression-decompression
Uncompressed small document	3647 bytes	521 ms	N/A
Compressed small document	886 bytes	126 ms	98 ms
Uncompressed large document	213160 bytes	30451 ms	N/A
Compressed large document	148269 bytes	21181 ms	1554 ms

(b) ***Table 13: Millau compression-decompression time compared to transmission time on a 56Kbits/s modem line: the compression-decompression and transmission of a Millau stream is faster than the transmission of an uncompressed stream.***

A typical eBusiness transaction is composed of a short request followed by a short answer. Without *Millau*, the transmission takes 1042 ms. With *Millau*, it takes 368 ms including the compression-decompression overhead giving a saving of 65% of the time. This can account for a significant portion on a large number of transactions. Note that the compression-decompression system has been implemented in Java. With an algorithm like GZIP which current implementation is in C, the compression-decompression time is smaller. We are working on a C implementation of the *Millau* compression-decompression algorithms which would allow a faster processing.

b) 6.2 *Millau* XML Remote Procedure Call

XML-RPC [12] is a very simple protocol for performing remote procedure calls over HTTP. It was designed by Userland Software, working with Microsoft. An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML. Procedure parameters can be scalars, numbers, strings, dates, etc., and can also be complex record and list structures. In our implementation, the body of the request is in *Millau* Binary XML.

The *Millau* encoding format appears to be particularly well adapted to the XML RPC because the exchanged messages are usually very short and may not contain redundant tags. So, as demonstrated in the previous section, the traditional compression algorithms have usually poor performances in term of compression rate with these kind of messages. On the other hand, XML-RPC uses a limited set of tags (20 different tags) and no attributes. All the tags can hold on one code page so there is no switch page. Actually, because *Millau* RPC has been designed to work with tokens, it is not necessary to store the code page in memory. Moreover, most of the content is of primitive type and *Millau* RPC can thus take advantage of the possibility of transmitting primitive types without text encoding. All these features make that the processing of XML-RPC requests and answers is efficient.

The *Millau* XML-RPC system is composed of a client which generates *Millau* requests and gets the answers in *Millau* format from the XML-RPC server and of a server which gets the requests for the clients, invokes the corresponding method (if found) and sends the response in *Millau* format or an error message (for example if the method is not found). The HTTP requests are first handled by a Java Servlet which passes the *Millau* body to the server.

The *init* method of the servlet creates the server which does all the work. It also registers the handler object which methods can be called from the client side. The server provides a *register* method to register handler object with a specific name and a *remove* method to remove a specific handler object from its name. References to the handlers are stored in a hash table on the server side. Each time the servlet receives a request, it passes the input stream to the server. The server parses the request encoded in *Millau* Binary XML-RPC format using a *Millau* BDOM parser. We are currently working on a version using a *Millau* BSAX parser instead of BDOM for improved efficiency. The server tries to find the handler object and the method corresponding to the method name of the request. If it finds it, it calls this method and encodes the response in *Millau* Binary XML-RPC format using the previously described *Millau* BDOM API. It then calls the method *writeBinaryXML* of this API which generates a *Millau* stream that can be sent as the body of the response.

To create a *Millau* XML-RPC client, the user passes the URL of a valid *Millau* XML-RPC server. Then the client can open a persistent connection with the server. The user can then call the *invoke* method of the client, passing the name of the method as "handler.method" and a Vector of the parameters. Details on the type mapping between XML-RPC and Java can be found in [13]. From the method name and the parameters, the client will generate a *Millau* XML-RPC request using the *Millau* BDOM API. Then, it invokes the *writeBinaryXML* method on the BDOM tree to generate a *Millau* stream that can be sent to the *Millau* XML-RPC server through the previously opened socket. The

client listens on the socket port to receive the response from the server. If it is a valid *Millau* XML-RPC response message, it is parsed and the result sent to the user; if it is an error message, the error is reported to the user. Notice that the error messages are also encoded in *Millau* XML-RPC format.

To evaluate the performance of this implementation, we made a benchmark which sends an array of 100 integers as a parameter and receives the same array as a return value. We compared the performances of our implementation with the Helma XML-RPC system [13]. The performance measure is given in table 14.

XML-RPC platform	Mean number of calls per second
<i>Millau</i> Binary XML-RPC	27 calls per second
Helma XML-RPC	12 calls per second

(a) **Table 14: Performance comparison between *Millau* XML-RPC and Helma XML-RPC**

Notice that HTTP may not be the best protocol to implement RPC because of the HTTP header which can be big compared to the Binary XML payload, especially if there are few parameters in the request. In case of very small requests, the performance improvement will not be significant because of the HTTP header overhead. A solution is to use persistent HTTP connections.

9. 7. Conclusion

As large number of XML documents are exchanged and streamed over the Internet medium, techniques for compact and efficient representation and exchange for this data become essential. In this paper, we describe a system called *Millau* for efficient encoding and streaming of XML structures. While traditional data compression algorithms lose the structure of the documents, *Millau* keeps the XML hierarchical structure. Moreover, *Millau* enables the separation of the content from the structure in order to be able to compress the text or multimedia data separately from the XML structure. This allows achieving better compression rates. This further allows an application to do some processing on the structure without having to download large volume of data. Moreover, our experiments show that, though traditional data compression algorithms are able to perform high compression rate on large XML files, they are much less effective towards small sized XML documents like the ones exchanged in eCommerce transactions. *Millau* achieves better compression rates for such documents.

To allow manipulation of *Millau* encoded XML documents at a layer transparent from the application, we provide both a SAX API and a DOM API conform to the standards of the Web. Additionally, we provide for both APIs methods for allowing applications to work directly with tokens instead of processing strings. Our experiments have shown that the processing of *Millau* tokens where up to five times faster than the processing of XML tags. We also developed data structures to efficiently map tags to tokens and to store the tokens. We provide algorithms to build the code spaces from the associated XML

13. Hannes Walln?, XML-RPC Library for Java, <http://helma.at/hannes/xmlrpc/>
14. Open Applications Group, <http://www.openapplications.org/>
15. Rob Barrett, Paul Maglio, J? Meyer, Steve Ihde, and Stephen Farrell, WBI Development Kit, <http://www.alphaworks.ibm.com/tech/wbidk>
16. Juan Ramon Velasco, Luis Alberto Velasco Lucia□ "Benefits of compression in HTTP applied to caching architectures", *Proceedings of the Third International WWW Caching Workshop*, 1998, <http://www.cache.ja.net/events/workshop/32/manchester.html>
17. Mark Nelson, *The Data Compression Book*, M&T Books, 1992
18. Ziv, J., and Lempel, A., "A universal algorithm for sequential data compression", *IEEE Transaction on Information Theory*, Volume 23, Number 3, May 1997, pages 337-343
19. Huffman, D.A., "A method for the construction of minimum-redundancy codes", *Proceedings of the IRE*, Volume 40, Number 9, September 1952, pages 1098-1101
20. Marc Girardot, "Efficient representation, streaming and exchange of XML content over the Internet medium", *Master Thesis, Eurecom Institute*, September 1999
21. R. Nigel Horspool, Gordon V. Cormack, "Constructing Word-Based Text Compression Algorithms", *IEEE Transaction on Information Theory*, 1992
22. N. Jesper Larsson, Alistair Moffat, "Offline Dictionary-Based Compression", *IEEE Transaction on Information Theory*, 1999
23. F. Yergeau, "UTF-8, a transformation format of ISO 10646", RFC 2279, Alis Technologies, January 1998, <http://www.ietf.org/rfc/rfc2279.txt>
24. IBM XML Parser for Java, <http://www.alphaworks.ibm.com/tech/xml4j>
25. Marc Girardot, Neel Sundaresan, "Efficient representation and streaming of XML content over the Internet medium", submitted to *IEEE International Conference on Multimedia and Expo 2000*

12. Vitae

Marc Girardot is a research intern at the IBM Almaden Research Center, California, USA. He holds a DEA honors degree in Networking and Distributed Applications from University of Nice, France and a Telecommunications Engineering degree (major: Multimedia Communications) from Ecole Nationale Supérieure des Télécommunications, Paris, France. His areas of research interests and expertise include multimedia, virtual reality, information theory, and Internet technologies with focus on XML.

Dr. Neel Sundaresan is a research manager of the eMerging Internet Technologies department at the IBM Almaden Research Center. He has been with IBM since December 1995 and has pioneered several XML and internet related research projects. He was one of the chief architects of the Grand Central Station project at IBM Research for building XML-based search engines. He received his PhD in CS in 1995. He has done research and advanced technology work in the area of Compilers and Programming Languages, Parallel and Distributed Systems and Algorithms, Information Theory, Data Mining and Semi-structured Data, Speech Synthesis, Agent Systems, and Internet Tools and Technologies. He has over 30 research publications and has given several invited and refereed talks and tutorials at national and international conferences. He has been a member of the W3C standards effort.

Appendix B: “xmlbin.doc” Specification

V. XML binary format

A. Goals

The primary goal of the XML binary format is to minimize the parsing and generation overhead of XML documents. It should be possible to take an arbitrary well-formed XML document in a text format, convert it to binary format and back without loss of fidelity.

The format is also designed to reduce the size of an XML document assuming that the text is left in the same character encoding. This is achieved by tokenizing the XML tag names and attribute names into variable sized numeric tokens.

In order to reduce XML text generation overhead when the original data is binary the format allows primitive types to be present in the stream in their ‘native’ binary representation. In order to simplify generating composite XML documents from existing data and fragments it also allows XML text embedded in the document. In this case the character encoding has to match the encoding of the main document.

Streaming access is really important both at parsing and generation time so the format is designed to allow incremental output of the XML document without forcing the creation of a global token table or string table at the beginning of the stream.

B. Format

In the following we are going to use the terms and definitions from the W3C Extensible Markup Language (XML) 1.0 recommendation (<http://www.w3.org/TR/REC-xml>). The structure of a binary XML document is the same as an XML text document. The XML markup structures (elements, comments, processing instructions, etc.) are represented by byte tokens. All XML ‘Name’ constructs are tokenized and replaced by name tokens. All tokens are fixed sized 8 bit unsigned integers.

Byte

`byte ::= 8 bit unsigned integer`

Token

`token ::= byte`

Here is a table of all the tokens. Each token defines what follows it. The most significant bit has special meaning depending on the type of the token. Token names will be used in production rules in uppercased bold letters.

TOKEN NAME	VALUE	DESCRIPTION	HIGH BIT
END	0	End of attribute list or element content	
ELEMENT	1	Element	Empty element
ATTRIBUTE	2	Attribute	
PI	3	Processing instruction	
COMMENT	4	Comment	
CDATA	5	CDATA section	
ENTITREF	6	Entity reference	Singleton
CHARREF	7	Character entity reference	Singleton
NAME	8	Name definition	
TEXTDATA	9	Character data	Singleton
XMLDATA	10	XML character data	
EXTENSION	11	XML structure intended for processor only	
BINARY	12	Binary data	Singleton
INT	13	Signed integer value	Singleton
UINT	14	Unsigned integer value	Singleton
DECIMAL	15	Decimal value	Singleton
MONEY	16	Currency value	Singleton
SMALLMONEY	17	Small range currency value	Singleton
FLOAT	18	4 byte floating precision value	Singleton
REAL	19	8 byte floating precision value	Singleton
DATETIME	20	8 byte date and time structure	Singleton
SMALLDATETIME	21	4 byte date and time structure	Singleton
GUID	22	16 byte UUID	Singleton
XMLDECL	23	XML PI	
DOCTYPEDECL	24	Document type declaration	

Variable sized unsigned integer values are represented by a multi-byte encoding format. This consists of a series of bytes where the most significant bit is a continuation flag. This allows 7 bit values per byte. The original value is encoded starting with the most significant 7 bits and continued big-endian order.

MultiByte

```
mb32 ::= byte+ /* 32 bit unsigned integer in multi-byte encoding */
```

Text data is represented as the number of characters encoded in a multi-byte value followed by the specified number of characters (using the 'Char' construct from the XML spec). White space can be preserved in the document by encoding it as TEXTDATA token followed by the textdata construct.

TextData

textdata ::= mb32 Char*

Named XML constructs (elements, PI-s, etc.) are encoded with the type token followed by a name token. The name token is an index into the name table. The name table is created by NAME tokens embedded in the XML stream.

NameDef

namedef ::= **NAME** TextData

Every 'namedef' creates a new entry in the name table and associates a unique unsigned integer id to it, which is simply the number of names in the table. These numbers are used as name tokens encoded in the multi-byte encoding format where a name XML construct is used.

The 'namedef' construct is allowed everywhere where a type token is allowed so it is possible to define all the names at the beginning of the stream. When a tokenizer operate on a 'normal' XML stream it will embed these as it encounters new names.

NameToken

nametoken ::= mb32

The following primitive binary data types are allowed in element content and attribute values in addition to character data: (ICS: need to define wire format here)

BinaryValue

```
value ::= int | /* 1, 2, 4, 8 byte signed integer */  
        uint | /* 1, 2, 4, 8 byte unsigned integer */  
        decimal | /* 5, 9, 13, 17 decimal type */  
        money | /* 8 byte currency */  
        smallmoney | /* 4 byte currency */  
        float | /* 4 byte floating precision number */  
        real | /* 8 byte floating precision number */  
        datetime | /* 8 byte structure */  
        smalldatetime | /* 4 byte structure */  
        guid /* 16 byte binary */
```

Arbitrary length binary data is encoded as the number of bytes encoded in a multi-byte value followed by the specified number of bytes.

BinaryData

```
bindata ::= mb32 byte*
```

Now let's go through the XML constructs in the recommendation to see what modifications are necessary for the binary format. The document construct stays the same.

Document

```
document ::= prolog element Misc*
```

Let's continue with the 'element' construct.

Element

```
element ::= EmptyElemTag  
          | STag content ETag
```

Elements with content are encoded by the ELEMENT token followed by optional attribute list and content closed by the END token.

Start-tag

```
STag ::= ELEMENT nametoken(Attribute)*  
Attribute ::= ATTRIBUTE nametoken AttValue  
ETag ::= END
```

In order to optimize singleton value elements (where there is only one construct in the content) if the most significant bit is set on the content type token it indicates that there is no need for an END token.

Content of Elements

```
content ::= (element | Data | Reference | CD Sect | PI | Comment)*
```

The element content can include typed data in addition to the original XML constructs so the original CharData is replaced by Data in the production.

Data


```

data ::= (TEXTDATA textdata) |
         (XMLDATA textdata) |
         (BINARY bindata) |
         (EXTENSION element) |
         (INT int) |
         (UINT uint) |
         (DECIMAL decimal) |
         (MONEY money) |
         (SMALLMONEY smallmoney) |
         (FLOAT float) |
         (REAL real) |
         (DATETIME datetime) |
         (SMALLDATETIME smalldatetime) |
         (GUID guid)

```

Empty elements are encoded by the ELEMENT token with the most significant bit set followed by the name token and optional attributes.

Empty Element

```
EmptyElemTag ::= ELEMENT+0x80 nametoken (Attribute)*
```

References are encoded as integer value for character references and as a name token for named references.

Entity Reference

```

Reference ::= EntityRef | CharRef
EntityRef ::= ENTITYREF nametoken
CharRef ::= CHARREF mb32

```

Processing instructions are encoded as name token for the target and text value for the characters in the PI body.

Processing Instructions

```
PI ::= PI nametoken textdata
```

Comments and CDATA sections are encoded as simple textdata following the token.

Comments

```
Comment ::= COMMENT nametoken textdata
```


CDATA Sections

`CD Sect ::= CDATA textdata`

Now back to the prolog and misc sections of the document production. The XMLDecl production always includes the version info as two bytes for major and minor version. This is followed by the encoding number according to the IANA assigned MIB number for the character set in multi-byte encoding. The last part is a byte indicating the standalone boolean value.

Prolog

```
prolog ::= XMLDecl? Misc* (doctypeddecl Misc*)?
XMLDecl ::= XMLDECL VersionInfo EncodingDecl SDDDecl
VersionInfo ::= Byte byte
EncodingDecl ::= mb32
SDDDecl ::= Byte
Misc ::= Comment | PI | S
```

The doctypeddecl part of the XML stream will always be left in XML text format.

Document Type Definition

[28] `doctypeddecl ::= DOCTYPEDECL textdata`

The use of the XMLDATA token allows mixing in arbitrary XML text, which of course has to be parsed by an XML parser. This is not true the other way around; the text cannot include a binary stream inside.

C. Possible extensions

It would be useful to allow tokenized data in content (both in attribute value and element content). We could add a new token type for referring to name tokens in the 'Data' production. This would help a lot in documents with heavy use of ID/IDREFS.

In addition we could introduce a content text table with tokenized values added similar to name tokens and again adding another token to the 'Data' production to refer to this table. Documents using lots of repeated attribute values are a good target for this.

D. Examples

In the examples below white space is ignored to make the encoding less verbose, the character count doesn't include it.

```
<a>
  <b>foo</b>
```

```

    <bar/>
</a>
<a>
    <b>text</b>
</a>

```

41 bytes

STREAM	DESCRIPTION
0x08	Name definition for 'a' gets name token 1
0x01 'a'	Textdata
0x01	Element
0x01	Name token 1 for 'a'
0x08	Name definition for 'b' gets name token 2
0x01 'b'	Textdata
0x01	Element
0x02	Name token 2 for 'b'
0x89	Text data token with singular bit set (END of 'b')
0x03 'f' 'o' 'o'	Textdata
0x08	Name definition for 'bar' gets name token 3
0x03 'b' 'a' 'r'	Textdata
0x81	Element with empty bit set
0x00	END of 'bar' (no attributes)
0x00	END of 'a'
0x01	Element
0x01	Name token 1 for 'a'
0x01	Element
0x02	Name token 1 for 'b'
0x89	Text data token with singular bit set (END of 'b')
0x04 't' 'e' 'x' 't'	Textdata
0x00	END of 'a'

34 bytes

```

<?xml version="1.0" ?>
<a:books xmlns:a="mybookschema" xmlns:dt="urn:schemas-microsoft-
com:datatypes">
    <a:book >
        <a:title>XML in action</a:title>
        <a:price dt:dt="fixed.14.4">34.2</a:price>
    </a:book>
    <a:book >
        <a:title>Data mining</a:title>
        <a:price dt:dt="fixed.14.4">50</a:price>
    </a:book>

```

</a:books>

293 bytes

STREAM	DESCRIPTION
0x17 0x01 0x00 0x6a 0x01	Xmldecl, 1.0, utf-8, standalone
0x08 0x07 'a:books'	Name definition for 'a:books' => 1
0x01 0x01	Element 'a:books'
0x08 0x07 'xmlns:a'	Name token 1 for 'xmlns:a' => 2
0x02 0x02	Attribute 'xmlns:a'
0x89 0x0c 'mybookschema'	Text data with singular bit set
0x08 0x08 'xmlns:dt'	Name token 1 for 'xmlns:a' => 3
0x02 0x03	Attribute 'xmlns:dt'
0x89 0x23 'urn:schemas-microsoft-com:datatypes'	Text data with singular bit set
0x08 0x06 'a:book'	Name definition for 'a:book' => 4
0x08 0x04	Element 'a:book'
0x08 0x07 'a:title'	Name definition for 'a:title' => 5
0x08 0x05	Element 'a:title'
0x89 0x0d 'XML in action'	Text data with singular bit set
0x08 0x07 'a:price'	Name definition for 'a: price'=> 6
0x08 0x06	Element 'a: price'
0x08 0x05 'dt:dt'	Name definition for 'dt:dt' => 7
0x02 0x07	Attribute 'dt:dt'
0x89 0x0a 'fixed.14.4'	Text data with singular bit set
0x90 [34.2]	8 byte currency data with singular bit set
0x00	END of 'a:book'
0x08 0x04	Element 'a:book'
0x08 0x05	Element 'a:title'
0x89 0x0b 'Data mining'	Text data with singular bit set
0x08 0x06	Element 'a: price'
0x02 0x07	Attribute 'dt:dt'
0x89 0x0a 'fixed.14.4'	Text data with singular bit set
0x90 [50]	8 byte currency data with singular bit set
0x00	END of 'a:book'
0x00	END of 'a:books'

212 bytes

E. Comparing to WAP Binary XML spec

Issues with the WAP spec:

It is not a complete XML document encoding, DTD, comments, named entity references are lost.

String table has to be at the beginning at the stream, doesn't allow incremental addition to the table.

There is no primitive data type encoding although it is possible to provide application specific binary data with the extension tags.

The element/attribute token space is very limited, it only allows 60 tokens per code page. This quickly makes it necessary to switch code pages around and so use 3 bytes per tag start.

It is not allowed to mix in XML text.

The main difference between the two solutions is the token space. In WAP the tokens include the name tokens as well. This limits the name token range but it can be switched. In this spec tokens can go up to 127 (1 bit is reserved for empty tag/singular content tag) and name tokens can be up to 32 bits.

In order to allow arbitrary extension construct (which can still be parsed by other parser) in this spec a special EXTENSION token is followed by an 'element' production so a whole XML sub-tree can be used to describe a processor specific structure. This is more flexible than the WAP solution with the three extension tokens allowing byte; integer and text extension but it might be overkill.

Appendix C:

Results of Analysis of the “xmlbin.doc” Specification

Analyzing xmlbin.doc

Document Name: xmlbin.doc

Path: C:\Documents and Settings\gplichta\Local Settings\Temporary Internet Files\OLK23C2

Document Format: Word Document

Compatibility Options: Microsoft Word 97

Built-in document properties:

Built-in Properties Containing Metadata: 3

Title: XML binary format

Author: Istvan Cseri

Company: Microsoft Corp.

Document Statistics:

Document Statistics Containing Metadata: 6

Creation Date: 10/27/1999 2:45:00 PM

Last Save Time: 10/28/1999 2:17:00 AM

Time Last Printed: [Blank]

Last Saved By: Istvan Cseri

Revision Number: 6

Total Edit Time (Minutes): 494 Minutes

Appendix D: Evidence of Diligence

This log shows activity from 12/15/99 (change # 49713 highlighted in bold below) up until 8/1/2000.

```
// File           : xmlxetxt.cpp -
//                               Defines implementation for
//                               - CXMLExecContext
//
// Written        : 03/15/99 (oliverse)
//
// History:
//
// @Version: Shiloh
// 64001 PXB 08/01/00 SQL64 - align esp stack
// 64498 ONS 06/21/00 FOR XML fix
// 63301 ONS 06/07/00 Perf
// 61502 ONS 05/15/00 For XML bugfix
// 61289 ONS 05/12/00 For XML XML-Data namespace error
// 60868 ONS 05/09/00 ForXML and OpenXML bugfixes
// 59822 ONS 05/02/00 For XML idrefs/nmtokens attributes combined with
//                subelements
// 59839 LRD 04/20/00 Shiloh RTM issues
// 59492 ONS 04/07/00 XMLData fix
// 59223 ONS 04/05/00 XMLDATA bugfix
// 55130 ONS 02/18/00 For XML fix 223312
// 54404 ONS 02/11/00 For XML fix
// 53395 ONS 02/09/00 For XML
// 53077 ONS 02/02/00 XML bugfix
// 53260 ONS 02/01/00 XML bug fixes
// 53078 ONS 01/28/00 XML bugfix
// 52963 ONS 01/27/00 FOR XML bug fixes
// 51674 ONS 01/15/00 XML Bugfixes
// 51507 ONS 01/14/00 FOR XML fixes
// 51158 ONS 01/12/00 For XML bug fixes
// 50405 ONS 01/05/00 XML Bug fixes
// 50152 ONS 12/28/99 XML Bug fixes
// 48293 ONS 12/17/99 FOR XML bug fixes
// 49713 ONS 12/15/99 Activate Tokenized XML support
// 47965 ONS 11/18/99 XML Bugfixes
```